Query Results over Ongoing Databases that Remain Valid as Time Passes By

Mülle, Yvonne ; Böhlen, Michael Hanspeter

Abstract: Ongoing time point now is used to state that a tuple is valid from the start point onward. For database systems ongoing time points have far-reaching implications since they change continuously as time passes by. State-of-the-art approaches deal with ongoing time points by instantiating them to the reference time. The instantiation yields query results that are only valid at the chosen time and get invalidated as time passes by. We propose a solution that keeps ongoing time points uninstantiated during query processing. We do so by evaluating predicates and functions at all possible reference times. This renders query results independent of a specific reference time and yields results that remain valid as time passes by. As query results, we propose ongoing relations that include a reference time attribute. The value of the reference time attribute is restricted by predicates and functions on ongoing attributes. We describe and evaluate an efficient implementation of ongoing data types and operations in PostgreSQL.

DOI: https://doi.org/10.1109/ICDE48307.2020.00127

Query Results over Ongoing Databases that Remain Valid as Time Passes By

Yvonne Mülle
Department of Computer Science
University of Zurich
Switzerland
Email: muelle@ifi.uzh.ch

Michael H. Böhlen
Department of Computer Science
University of Zurich
Switzerland
Email: boehlen@ifi.uzh.ch

***Abstract*—Ongoing time point now is used to state that a tuple is valid from the start point onward. For database systems ongoing time points have far-reaching implications since they change continuously as time passes by. State-of-the-art approaches deal with ongoing time points by instantiating them to the reference time. The instantiation yields query results that are only valid at the chosen time and get invalidated as time passes by.**

We propose a solution that keeps ongoing time points *uninstantiated* during query processing. We do so by evaluating predicates and functions at all possible reference times. This renders query results independent of a specific reference time and yields results that remain valid as time passes by. As query results, we propose *ongoing relations* that include a *reference time attribute*. The value of the reference time attribute is restricted by predicates and functions on ongoing attributes. We describe and evaluate an efficient implementation of ongoing data types and operations in PostgreSQL.

I. Introduction

Data that are associated with a valid time interval [1] are present in real-world applications that deal with employment contracts, insurance policies, software bugs, etc. The ongoing time point now is commonly used to state that the contract, policy, bug, etc. is valid from the start point onward.

The ongoing time point now changes its value when time passes by and the reference time is used to determine the value. At each reference time, now instantiates to the time point equal to the reference time. For example, at reference time $08/15$, now instantiates to time point $08/15$ and at reference time $08/16$, it instantiates to time point $08/16$. Throughout the paper, we use time points in the mm/dd format relative to 2019: time point $08/15$ denotes August 15, 2019.

A key assumption of database systems is that query results only get outdated if data is modified explicitly. This happens if data is inserted, updated, or deleted. The assumption no longer holds if now is stored in the database or when queries are evaluated on databases with ongoing time points [2]–[4]. In this case, query results get also outdated as a result of time passing by. This has significant drawbacks. First, query results, including materialized views and cached query results, must be re-computed before they can be accessed. Second, because ongoing time points are replaced by fixed time points, it is impossible for applications to identify result time points that change when time passes by.

This paper proposes an elegant and efficient solution that preserves ongoing time points in query results and that evaluates queries at all possible reference times to get results that remain valid as time passes by. Formally, given a database D with ongoing time points and a query Q, we want to compute a query result $Q(D)$, such that at every possible reference time rt, the query result is equivalent to the result obtained by instantiating now in D and evaluating the query on the instantiated database: $\forall rt(\|Q(D)\|_{rt} \equiv Q(\|D\|_{rt}))$. The bind operator $\|\cdot\|_{rt}$ replaces all occurrences of now with the reference time rt.

To support queries with predicates and functions on ongoing attributes, the key challenges are (1) the evaluation of queries to results that remain valid as time passes by and (2) the representation of these results.

To get results that remain valid, we keep ongoing time points uninstantiated. We define six core operations predicate $<$, functions $\min$ and $\max$, and the logical connectives $\wedge$, $\vee$, $\neg$. At each reference time, their results are equal to the results obtained by the corresponding operations for fixed data types on the instantiated input arguments. We provide equivalences for the core operations and for additional operations that are expressed with the core operations. The equivalences are used for an efficient implementation. We represent the results of predicates and logical connectives as *ongoing booleans*, i.e., booleans whose truth value depends on the time. The results of relational algebra operators are represented as *ongoing relations* that include a reference time attribute RT. The value of RT includes the reference times when a tuple belongs to the instantiated relations. The reference time of a tuple is restricted by predicates in queries. We represent the value of the RT attribute with a finite set of fixed time intervals. Thus, only predicates that evaluate to booleans that change their value a finite number of times are allowed. The tuples in base ongoing relations have a trivial reference time, i.e., $RT = \{(-\infty, \infty)\}$. Tuples with an empty reference time, i.e., $RT = \{\}$, are deleted.

Our technical contributions are the following:

- We propose the ongoing time domain Ω for *ongoing time points*. The time domain is closed for $\min$ and $\max$, i.e., the evaluation of $\min$ and $\max$ on Ω again yields an ongoing time point of Ω.

- We define predicates, functions and logical connectives that keep ongoing time points uninstantiated during query processing.
- We introduce *ongoing relations* with a reference time attribute to represent query results that remain valid as time passes by. The value of the RT attribute is set by the database system and restricted by predicates on ongoing attributes.
- We define the relational algebra for ongoing relations. The result of each operator is an ongoing relation that remains valid as time passes by.
- We describe an efficient implementation of ongoing data types and operations on these data types in the kernel of PostgreSQL.

The paper is organized as follows. Section II introduces our running example. Section III discusses related work. Section IV provides preliminaries. We define the time domain for ongoing time points in Section V. Predicates and functions on ongoing time points and time intervals whose results remain valid are discussed in Section VI. Section VII introduces ongoing relations and defines a relational algebra on them. Section VIII discusses the implementation of our solution in PostgreSQL. The evaluation is described in Section IX. Section X concludes the paper and points to future research.

II. Running Example

Consider a company that keeps track of bugs associated with the individual components of its email service. Prioritized bugs have fixed start points that indicate when the bug was discovered and fixed end points that indicate the deadline for resolving the bug internally. Deprioritized bugs have fixed start points but end points that keep increasing. These end points are *ongoing*. A bug is open iff it has been discovered but not yet resolved internally. Once a bug has been resolved internally, its fix will be deployed in a future patch to the production servers. The patches for the components of the email service are pre-scheduled. Selected relations of our running example are shown in Fig. 1 and discussed below.

B

	BID	C	VT	RT
b_1	500	Spam filter	$[01/25, now)$	$\{(-\infty, \infty)\}$
b_2	501	Spam filter	$[03/30, 08/21)$	$\{(-\infty, \infty)\}$

P

	PID	C	VT	RT
p_1	201	Spam filter	$[08/15, 08/24)$	$\{(-\infty, \infty)\}$
p_2	202	Spam filter	$[08/24, 08/27)$	$\{(-\infty, \infty)\}$

L

	Name	C	VT	RT
l_1	Ann	Spam filter	$[01/20, 08/18)$	$\{(-\infty, \infty)\}$
l_2	Bob	Spam filter	$[08/18, now)$	$\{(-\infty, \infty)\}$

Fig. 1: Relations with ongoing time points.

Relation $\mathbf{B}$ illustrates bugs described by identifier *BID*, the name of the affected component C, the valid time interval VT during which the bug is open, and the reference time *RT* when the tuple belongs to the instantiated relations (cf. below and Section VII for the details). For instance, tuple b_1 records deprioritized bug 500 for the Spam filter component that has been open from 01/25 until now.

Relation $\mathbf{P}$ illustrates patches described by patch number *PID*, component *C* to which the patch applies, valid time interval *VT* during which the patch is live, and the reference time *RT*. For instance, tuple p_1 states that patch 201 of the Spam filter is live from 08/15 until 08/24 exclusively.

Relation $\mathbf{L}$ lists the technical leads. A technical lead is described by their name, component *C* they are responsible for, valid time interval *VT* during which they are responsible for the component, and the reference time *RT*. For instance, tuple l_2 records that Bob is the technical lead for the Spam filter component from 08/18 until now.

Relations $\mathbf{B}$, $\mathbf{P}$, and $\mathbf{L}$ are *base ongoing relations*. All tuples belong to the instantiated relations at all reference times and have a trivial reference time, i.e., $RT = \{(-\infty, \infty)\}$. The reference time is restricted by predicates on ongoing attributes. We will discuss the restriction of a tuple's reference time in the following.

To schedule bug fixes, reprioritize bugs, and assess unresolved bugs, we run a query that joins bugs that affect the Spam filter with upcoming patches and technical leads:

$$\begin{aligned}\mathbf{V} \leftarrow \pi_{BID,\mathbf{B}.VT,PID,\text{Name},\mathbf{B}.VT \cap \mathbf{L}.VT}(\\ \sigma_{C=\text{'Spam filter'}}(\mathbf{B}) \\ \bowtie_{(\mathbf{B}.C=\mathbf{P}.C) \wedge (\mathbf{B}.VT \text{ before } \mathbf{P}.VT)} \mathbf{P} \\ \bowtie_{(\mathbf{B}.C=\mathbf{L}.C) \wedge (\mathbf{B}.VT \text{ overlaps } \mathbf{L}.VT)} \mathbf{L})\end{aligned}$$

We illustrate the computation of the reference time RT for $b_1 \bowtie_\theta p_1$ with $\theta = ((\mathbf{B}.C = \mathbf{P}.C) \wedge (\mathbf{B}.VT \text{ before } \mathbf{P}.VT))$. Conceptually, all occurrences of now in predicate $\theta(b_1, p_1)$ are replaced with each possible reference time rt in turn and the predicate is evaluated. This yields the following results for the *before* predicate:

rt	$[01/25, now)$	$[08/15, 08/24)$	$b_1.VT$ before $p_1.VT$
...	...	...	...
08/14	$[01/25, 08/14)$	$[08/15, 08/24)$	*true*
08/15	$[01/25, 08/15)$	$[08/15, 08/24)$	*true*
08/16	$[01/25, 08/16)$	$[08/15, 08/24)$	*false*
...	...	...	...

At all reference times when the join predicate evaluates to *true*, the result tuple belongs to the instantiated relations. In our example these are all reference times from 01/26 up to 08/15 and we get $RT = \{[01/26, 08/16)\}$.

V

	BID	B.VT	PID	Name	B.VT ∩ L.VT	RT
v_1	500	$[01/25, now)$	201	Ann	$[01/25, +08/18)$	$\{[01/26, 08/16)\}$
v_2	500	$[01/25, now)$	202	Ann	$[01/25, +08/18)$	$\{[01/26, 08/25)\}$
v_3	500	$[01/25, now)$	202	Bob	$[08/18, now)$	$\{[08/19, 08/25)\}$
v_4	501	$[03/30, 08/21)$	202	Ann	$[03/30, 08/18)$	$\{(-\infty, \infty)\}$
v_5	501	$[03/30, 08/21)$	202	Bob	$[08/18, +08/21)$	$\{[08/19, \infty)\}$

Fig. 2: Query result $\mathbf{V}$ remains valid as time passes by.

Query result $\mathbf{V}$ includes the tuples illustrated in Fig. 2. Note that (1) all ongoing time points are preserved in $\mathbf{V}$. For instance, the value of the $\mathbf{B}.VT$ attribute makes it possible to

identify prioritized and deprioritized bugs. (2) The intersection $\mathbf{B}.VT \cap \mathbf{L}.VT$ states when a technical lead is responsible for a bug. Consider tuple v_1 with $b_1.VT \cap l_1.VT = [01/25, +08/18)$, which is an ongoing time interval. Tuple v_1 states that Ann is the responsible technical lead for bug 500 from $01/25$ until possibly earlier, but not later than $08/17$. Clearly, fixed time points together with now are not sufficient to represent such results. (3) The reference time of a tuple is restricted by predicates on ongoing attributes. For each operator, the reference time of the result tuples is determined by the reference times when the input tuples belong to the instantiated relations *and* the reference times when the predicate evaluates to *true*. The reference time of the input tuples is relevant since it is the result of predicates in earlier operators that derive these tuples. For instance, the reference time of the result tuples of join $\sigma_{C='\text{Spam filter}'}(\mathbf{B}) \bowtie_\theta \mathbf{P}$ was restricted by join predicate θ. These tuples are then input tuples for the join with ongoing relation $\mathbf{L}$.

III. Related Work

The most commonly used ongoing time point is now. The state-of-the-art approach to deal with ongoing time points is to instantiate them, i.e., replace them with the reference time. Commercial database systems use the compile time as the reference time whereas research approaches usually use the evaluation time as the reference time. Below we discuss the implications of both choices for storing ongoing time points, query processing, and the validity of query results.

Existing database systems cannot store ongoing time points. They instantiate ongoing time points immediately at compile time when statements are issued. The SQL-92 standard [5] includes the reserved keywords CURRENT_TIME, CURRENT_DATE, and CURRENT_TIMESTAMP that denote the ongoing time point now for different time granularities. These constructs can be used in SQL statements, but are instantiated immediately at compile time.

Various research approaches have progressed the basic solution offered by commercial database systems. The key idea is to store ongoing time points and instantiate them when accessing the data during query processing. The advantage of instantiating ongoing time points is that existing query processing techniques can be used since the instantiation eliminates ongoing time points [6]–[12]. The disadvantage is that query results are only valid at the chosen reference time and get outdated by time passing by. Below we discuss different aspects of the instantiation that have been investigated [2], [13]–[15]. Throughout, we use $\mathcal{T}$ to denote the domain of fixed time points.

Clifford et al. [2], [16] propose a solution that handles ongoing time point now during query processing. Their framework instantiates now whenever it is accessed. Thus, queries are evaluated on instantiated relations without ongoing time points. This yields result relations that are only valid at the time when now was accessed.

Anselma et al. [4] propose an algebra for relations with ongoing time points. Their goal is an approach that copes with four commonly used representations of now: *Min*, *Max* [17], *Null*, and *Empty Range* [18], [19]. Their time domain is $\mathcal{T} \cup \{now\}$. They introduce intersection and difference functions that may keep ongoing time points uninstantiated. For instance, ongoing time points are not instantiated when the resulting time interval contains now as end point like in $[10/14, now) \cap [10/17, now) = [10/17, now)$. Their approach must instantiate now for more complex end points. For instance, $[10/17, 10/22) \cap [10/17, now) = [10/17, 10/20)$ at reference time $10/20$. Anselma et al. [20] have extended their approach to support indeterminacy for tuples with now. They have not worked out how predicates on ongoing time points are defined and evaluated.

Snodgrass [21] proposes *Forever* instead of the ongoing time point now. *Forever* denotes the largest time point in the time domain, which is a fixed time point. Existing query evaluation approaches for relations without ongoing time points can be used on relations that use *Forever*. However, replacing now with *Forever* leads to incorrect results. For instance, at reference time $05/14$ the query *"Which bugs might be resolved before patch 201 goes live?"* is not answered correctly. Evaluating the query on relations $\mathbf{B}$ and $\mathbf{P}$ of Fig. 1 with *Forever* as the end point results in bug 500 not being part of the result relation, which is not correct.

Torp et al. [3] propose a solution for modifications of temporal databases. They show that performing temporal modifications on tuples that are instantiated when accessed leads to incorrect modifications and thus, incorrect data in the database. The authors propose time domain $\mathcal{T}_f = \mathcal{T} \cup \{\min(a, now) | a \in \mathcal{T}\} \cup \{\max(a, now) | a \in \mathcal{T}\}$ to handle such modifications. Instead of now, they use the minimum and maximum of a time point and now to correctly modify the database. Time domain T_f supports intersection and difference functions that do not instantiate ongoing time points. Torp et al. use these two functions to express temporal modifications that remain valid as time passes by. Their approach cannot evaluate predicates on uninstantiated time attributes. Queries with such predicates resort to Clifford's approach. Thus, query results get invalidated by time passing by.

Moving objects [22] change their spatial position as time advances. Research approaches in this area deal with different types of queries on moving objects: static queries [23], [24], continuous queries [25]–[28], and time-parametrized queries [29]. *Static queries* instantiate the moving objects at a given reference time and are evaluated at fixed spatial positions. These approaches are similar to the approach of Clifford et al. [2], which instantiates ongoing time points. *Continuous queries* compute results that remain valid for a short time span, e.g., 10 seconds, before the query is re-evaluated. The results are continuously returned to applications. A query result contains pairs of moving object(s) and the reference times when the pair belongs to the result. Structurally, the query result is similar to ongoing relations with a reference time attribute. However, the result of a continuous query is only valid for a short time span and gets invalidated by time passing by. *Time-parametrized queries* [29] incrementally

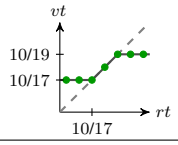
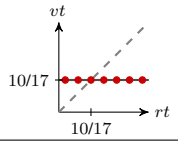
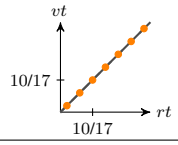
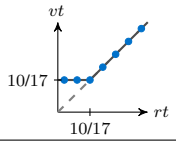
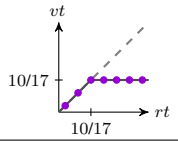
	Ongoing time point	Type			
		Fixed time point	Time point now	Growing time point	Limited time point
Notation					
- as a+b	a+b	a+a	$-\infty$+∞	a+∞	$-\infty$+b
- short	a+b	a	now	a+	+b
Meaning	not earlier than a, but not later than b	time point a	the *current* time point	not earlier than a, possibly later	possibly earlier, but not later than b
Example	10/17+10/19	10/17	now	10/17+	+10/17
Semantics					

Fig. 3: Illustration of ongoing time points a+b.

determine their results. The result consists of three parts: the objects that satisfy the spatial query, the reference time until when the result is valid, and the objects that change the result. The result is only valid from the time when the query was issued until the returned reference time.

Now-relative and indeterminate time points have been proposed as extensions of ongoing time point now [2]. A now-relative time point, e.g., $now + 5$ days, shifts now by 5 days into the future. An indeterminate time point specifies a period during which an event will occur. For instance, the indeterminate time point 04/17 $\sim$ 04/20 as the end point of a resolved bug states that the resolution occurred sometime between 04/17 and 04/20. These extensions are orthogonal to our generalization of now.

IV. Preliminaries

We assume a linearly ordered, discrete *time domain* $\mathcal{T}$ with $-\infty$ as the lower limit and ∞ as the upper limit. A *time point* is an element of time domain $\mathcal{T}$. A *time interval* $[t_s, t_e)$ consists of an inclusive start point t_s and an exclusive end point t_e. *Fixed* data types consist of values that do not change as time passes by. Examples are integers, strings, booleans, and time points of $\mathcal{T}$. *Ongoing* data types include values that change as time passes by. Ongoing values can be *instantiated* to fixed values. We consider the following ongoing data types: ongoing time points, ongoing booleans, and composite structures (intervals, tuples, relations) that include ongoing time points. The bind operator $\|x\|_{rt}$ performs the instantiation of x at reference time $rt \in \mathcal{T}$. If x is composite each component is instantiated. We use the ^{F}-superscript for operations on fixed data types. For instance, $\min^F$ is the standard minimum function over fixed arguments, i.e., $\min^F(j,k) = j$ if $j < k$ and $\min^F(j,k) = k$ otherwise.

$R = (\mathbf{A})$ denotes the schema of a fixed relation $\mathbf{R}$ with fixed attributes $\mathbf{A} = A_1, \ldots, A_n$. A tuple r with schema R is a finite list that contains for every A_i a value from the domain of A_i. A relation $\mathbf{R}$ over schema R is a finite set of tuples over R. $r.A_i$ denotes the value of attribute A_i in tuple r. $\theta(r)$ denotes the application of predicate θ to tuple r. An *ongoing relation* is a relation with fixed and ongoing attributes $\mathbf{A}$ and a reference time attribute RT (cf. Definition 5). The value of RT is a set of fixed time intervals.

Valid time [30], transaction time [30], and reference time are separate concepts. Consider a tuple b that refers to bug 500 with valid time $VT = [01/25, now)$, transaction time $TT = [01/26, now)$, and reference time $RT = \{[03/15, \infty)\}$. The valid time states when a tuple is valid in the real world: bug 500 is open from 01/25 until now. The valid time is set by the user. The transaction time states when a tuple was modified in the relation: tuple b was inserted in 01/26 and not modified since. The transaction time is restricted by the database system through database modifications, i.e., insert, update, and delete statements. The reference time states when a tuple belongs to the instantiated relations: tuple b belongs to the instantiated relations from 03/15 on. The reference time is set by the database system and restricted by the predicates on ongoing attributes in queries.

V. Ongoing Time Data Types

This section defines the ongoing time domain Ω, ongoing time points, and ongoing time intervals. In contrast to previously proposed ongoing time domains, Ω is closed for minimum and maximum functions.

A. Ongoing Time Points

Definition 1 (Ongoing Time Domain Ω): Let $\mathcal{T}$ be the time domain of fixed time points. Ongoing time domain Ω consists of all possible ongoing time points a+b:

$$\Omega = \{a\text{+}b \mid \exists a, b \in \mathcal{T}(a \leq b)\}$$

The intuitive meaning of the ongoing time point a+b is *not earlier than a, but not later than b*. For instance, 10/17+10/19 means *not earlier than* 10/17*, but not later than* 10/19.

Definition 2 (Ongoing Time Point): Let $rt \in \mathcal{T}$ be a reference time and $a, b \in \mathcal{T}$ with $a \leq b$. The ongoing time point a+b is defined as

$$\|a\text{+}b\|_{rt} = \begin{cases} a & rt \leq a \\ rt & a < rt < b \\ b & \text{otherwise} \end{cases}$$

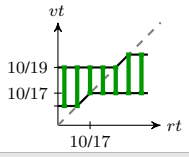
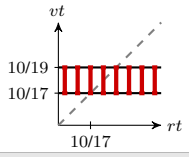
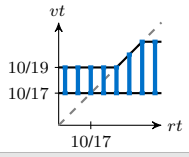
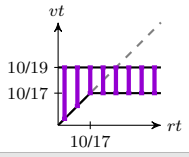
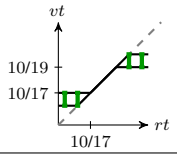
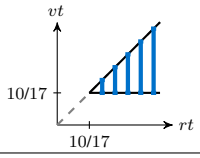
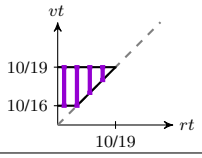
	Ongoing time interval	Type		
		Fixed time interval	**Expanding time interval**	**Shrinking time interval**
non-empty	if $a \leq b < c \leq d$	if $a = b < c = d$	if $a = b < c < d$	if $a < b < c = d$
	$[10/16{+}10/17, 10/19{+}10/20)$	$[10/17, 10/19)$	$[10/17, 10/19{+}10/21)$	$[{+}10/17, 10/19)$
				
partially empty	if $a < c \leq b$ or $c \leq a \leq b < d$	never	if $c \leq a \leq b < d$	if $a < c \leq d \leq b$
	$[10/16{+}10/19, 10/17{+}10/20)$	–	$[10/17, now)$	$[10/16{+}, 10/19)$
				

Fig. 4: Illustration of ongoing time intervals $[a{+}b, c{+}d)$.

For instance, ongoing time point 10/17+10/19 instantiates to time point 10/17 up to reference time 10/17. Between reference times 10/17 and 10/19 the ongoing time point instantiates to the reference time. Afterwards, it instantiates to time point 10/19.

A *fixed* time point a, current time point now, a *growing* time point $a{+}$, and a *limited* time point ${+}b$ can all be expressed as ongoing time points of the form $a{+}b$. This is illustrated in Fig. 3. For instance, fixed time point $a = a{+}a$ is an ongoing time point that instantiates to time point a at all reference times; time point $now = -\infty{+}\infty$ is an ongoing time point that instantiates to the reference time at all reference times.

Table I summarizes the properties of time domains $\mathcal{T}$, $\mathcal{T}_{now} = \mathcal{T} \cup \{now\}$ [2], $\mathcal{T}_f = \mathcal{T} \cup \{\min(a, now) | \ a \in \mathcal{T}\} \cup \{\max(a, now) | \ a \in \mathcal{T}\}$ [3], and Ω. For each time domain we show if it includes fixed or ongoing time points and if it is closed for $\min$ and $\max$.

TABLE I: Properties of time domains.

Time Domain	Fixed	Ongoing	Closed
$\mathcal{T}$	yes	no	yes
$\mathcal{T}_{now}$	yes	yes	no
$\mathcal{T}_f$	yes	yes	no
Ω	**yes**	**yes**	**yes**

B. Ongoing Time Intervals

An ongoing time interval $[t_s, t_e)$ is a closed-open time interval with domain $\Omega \times \Omega$. As an example, time interval $[10/17, now)$ is an ongoing time interval. An ongoing time interval can be instantiated to a fixed time interval by instantiating start and end points:

$$\forall rt \in \mathcal{T}(\|[t_s, t_e)\|_{rt} = [\|t_s\|_{rt}, \|t_e\|_{rt}))$$

The ongoing time interval $[a{+}b, c{+}d)$ generalizes *fixed* time intervals, *expanding* time intervals, and *shrinking* time intervals. Their semantics are illustrated in Fig. 4. For instance, an expanding time interval instantiates to time intervals whose duration increases with increasing reference time. The duration can increase for all reference times or up to a certain reference time. An example for the first case is ongoing time interval $[10/17, now)$ with $d = \infty$. An example for the latter case is ongoing time interval $[10/17, 10/19{+}10/21)$ with $d = 10/21$. It instantiates to time intervals with increasing duration up to reference time 10/21. From reference time 10/21 on, it instantiates to time interval $[10/17, 10/21)$.

An ongoing time interval can be partially empty. A partially empty time interval instantiates to empty time intervals at some reference times and to non-empty time intervals at others. This is illustrated in Fig. 4. For instance, ongoing time interval $[10/17, now)$ instantiates to empty time intervals up to reference time 10/17. At these reference times, end point now instantiates to time points that are less than or equal to start point 10/17 and the interval is empty. Afterwards, now instantiates to time points greater than 10/17 and $[10/17, now)$ instantiates to non-empty time intervals.

VI. Operations on Ongoing Data Types

This section defines operations, i.e., functions, predicates, and logical connectives, on ongoing time data types whose results remain valid as time passes by. At each reference time, their results are equal to the results obtained by the corresponding operation on fixed data types. We provide and prove equivalences for our six core operations $<, \min, \max, \wedge, \vee, \neg$ and show how we use these core operations in equivalences for additional operations on ongoing data types.

Since ongoing time points and time intervals instantiate to different values depending on the reference time the truth value of predicates depends on the reference time. To represent their result, we use *ongoing booleans* whose boolean value depends on the reference time.

Definition 3 (Ongoing Boolean): Let $rt \in \mathcal{T}$ be a reference time. Let $S_t \subseteq \mathcal{T}$ and $S_f \subseteq \mathcal{T}$ be disjoint subsets of all possible reference times with $S_t \cup S_f = \mathcal{T}$. The ongoing

boolean $\mathbf{b}[S_t, S_f]$ is defined as

$$\|\mathbf{b}[S_t, S_f]\|_{rt} = \begin{cases} \text{true} & rt \in S_t \\ \text{false} & rt \in S_f \end{cases}$$

An ongoing boolean $\mathbf{b}[S_t, S_f]$ is *true* at the reference times in S_t and *false* at the reference times in S_f. For instance, ongoing boolean $\mathbf{b}[\{[10/18, \infty)\}, \{(-\infty, 10/18)\}]$ is *true* at reference time 10/18 (as well as at all later reference times), and it is *false* at the reference times earlier than 10/18. Ongoing booleans generalize booleans. Boolean *true* is equivalent to ongoing boolean $\mathbf{b}[\{(-\infty, \infty)\}, \emptyset]$, which is *true* at all reference times. Boolean *false* is equivalent to ongoing boolean $\mathbf{b}[\emptyset, \{(-\infty, \infty)\}]$. The generalization makes it possible to combine predicates that evaluate to booleans with predicates that evaluate to ongoing booleans in logical expressions.

Definition 4 (Core Operations): Let $t_1, t_2, t \in \Omega$ be ongoing time points. Let $\mathbf{b}_1, \mathbf{b}_2, \mathbf{b} \in \Gamma$ be ongoing booleans. The core operations on ongoing data types are defined as follows:

Operation	Definition
$<$	$t_1 < t_2 = \mathbf{b}$ iff $\forall rt \in \mathcal{T}(\|\mathbf{b}\|_{rt} \Leftrightarrow \|t_1\|_{rt} <^F \|t_2\|_{rt})$
min	$\min(t_1, t_2) = t$ iff $\forall rt \in \mathcal{T}(\|t\|_{rt} = \min^F(\|t_1\|_{rt}, \|t_2\|_{rt}))$
max	$\max(t_1, t_2) = t$ iff $\forall rt \in \mathcal{T}(\|t\|_{rt} = \max^F(\|t_1\|_{rt}, \|t_2\|_{rt}))$
$\wedge$	$\mathbf{b}_1 \wedge \mathbf{b}_2 = \mathbf{b}$ iff $\forall rt \in \mathcal{T}(\|\mathbf{b}\|_{rt} \Leftrightarrow \|\mathbf{b}_1\|_{rt} \wedge^F \|\mathbf{b}_2\|_{rt})$
$\vee$	$\mathbf{b}_1 \vee \mathbf{b}_2 = \mathbf{b}$ iff $\forall rt \in \mathcal{T}(\|\mathbf{b}\|_{rt} \Leftrightarrow \|\mathbf{b}_1\|_{rt} \vee^F \|\mathbf{b}_2\|_{rt})$
$\neg$	$\neg\mathbf{b}_1 = \mathbf{b}$ iff $\forall rt \in \mathcal{T}(\|\mathbf{b}\|_{rt} \Leftrightarrow \neg^F\|\mathbf{b}_1\|_{rt})$

An operation on ongoing data types evaluates to a result that, at each reference time, is equal to the result obtained by the corresponding operation on fixed data types. This yields results that remain valid as time passes by.

All other predicates and functions on ongoing data types are defined analogously.

Example 1: Consider min for ongoing time points and the corresponding function $\min^F$ for fixed time points. The result of $\min(10/17, now)$ is ongoing time point $t = +10/17$ (cf. Theorem 1). At each reference time, it is equal to the time point obtained from evaluating $\min^F$ on the instantiated input arguments, i.e., $+10/17$ is equal to $\min^F(\|10/17\|_{rt}, \|now\|_{rt})$ at every reference time rt. Fig. 5 illustrates the equality for reference times 10/15 and 10/19.

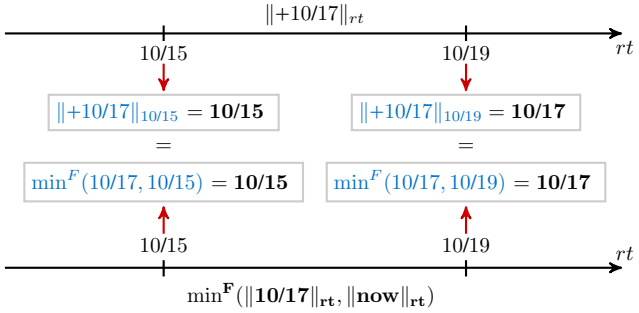


Fig. 5: The result of min remains valid.

Theorem 1: Let $a{+}b, c{+}d \in \Omega$ be ongoing time points. Let $\mathbf{b}[S_t, S_f], \mathbf{b}[\tilde{S}_t, \tilde{S}_f] \in \Gamma$ be ongoing booleans. The results of the operations on ongoing data types given in Definition 4 are equivalent to the following ongoing values:

Operation	Equivalence		
$<$	$a{+}b < c{+}d \equiv$		
	$\mathbf{b}[\{(-\infty, \infty)\}, \emptyset]$	$a \le b < c \le d$	(1)
	$\mathbf{b}[\{(-\infty, c)\}, \{[c, \infty)\}]$	$a < c \le d \le b$	(2)
	$\mathbf{b}[\{[b+1, \infty)\}, \{(-\infty, b+1)\}]$	$c \le a \le b < d$	(3)
	$\mathbf{b}[\{(-\infty, c), [b+1, \infty)\}, \{[c, b+1)\}$	$a < c \le b < d$	(4)
	$\mathbf{b}[\emptyset, \{(-\infty, \infty)\}]$	otherwise	(5)
min	$\min(a{+}b, c{+}d) \equiv \min^F(a, c){+}\min^F(b, d)$		
max	$\max(a{+}b, c{+}d) \equiv \max^F(a, c){+}\max^F(b, d)$		
$\wedge$	$\mathbf{b}[S_t, S_f] \wedge \mathbf{b}[\tilde{S}_t, \tilde{S}_f] \equiv \mathbf{b}[S_t \cap^F \tilde{S}_t, S_f \cup^F \tilde{S}_f]$		
$\vee$	$\mathbf{b}[S_t, S_f] \vee \mathbf{b}[\tilde{S}_t, \tilde{S}_f] \equiv \mathbf{b}[S_t \cup^F \tilde{S}_t, S_f \cap^F \tilde{S}_f]$		
$\neg$	$\neg\mathbf{b}[S_t, S_f] \equiv \mathbf{b}[S_f, S_t]$		

The proof of Theorem 1 is provided in the extended online version [31].

We use our core operations to provide equivalences for predicates and functions on ongoing time points and time intervals. Table II illustrates the equivalences for selected predicates and functions. For instance, the intersection $[t_s, t_e) \cap [\tilde{t}_s, \tilde{t}_e)$ on ongoing time intervals is equivalent to the ongoing time interval $[\max(t_s, \tilde{t}_s), \min(t_e, \tilde{t}_e))$. Equivalences for additional predicates are given in the extended online version [31].

TABLE II: Equivalences for predicates and function on ongoing time points and time intervals.

Operation	Equivalence
$\le$	$t_1 \le t_2 \equiv \neg(t_2 < t_1)$
	Example $now \le 10/17 = \neg(\mathbf{b}[\{[10/18, \infty)\}, \{(-\infty, 10/18)\}])$ $= \mathbf{b}[\{(-\infty, 10/18)\}, \{[10/18, \infty)\}]$
$=$	$t_1 = t_2 \equiv t_1 \le t_2 \wedge t_2 \le t_1$
	Example $(10/17 = now)$ $= \mathbf{b}[\{[10/17, 10/18)\}, \{(-\infty, 10/17), [10/18, \infty)\}]$
before	$[t_s, t_e)$ before $[\tilde{t}_s, \tilde{t}_e) \equiv t_e \le \tilde{t}_s \wedge t_s < t_e \wedge \tilde{t}_s < \tilde{t}_e$
	Example $[10/17, now)$ before $[10/20, 10/25)$ $= \mathbf{b}[\{[10/18, 10/21)\}, \{(-\infty, 10/18), [10/21, \infty)\}]$
overlaps	$[t_s, t_e)$ overlaps $[\tilde{t}_s, \tilde{t}_e) \equiv t_s < \tilde{t}_e \wedge \tilde{t}_s < t_e \wedge t_s < t_e \wedge \tilde{t}_s < \tilde{t}_e$
	Example $[10/17, now)$ overlaps $[10/14, 10/20)$ $= \mathbf{b}[\{[10/18, \infty)\}, \{(-\infty, 10/18)\}]$
$\cap$	$[t_s, t_e) \cap [\tilde{t}_s, \tilde{t}_e) \equiv [\max(t_s, \tilde{t}_s), \min(t_e, \tilde{t}_e))$
	Example $[10/17, now) \cap [10/14, 10/20) = [10/17, +10/20)$

For predicates on ongoing time intervals we must explicitly consider the non-emptiness of the ongoing time intervals. For instance, the overlaps predicate is equivalent to the ongoing boolean that results from the usual overlaps check $t_s < \tilde{t}_e \wedge \tilde{t}_s < t_e$ and an explicit non-empty check $t_s < t_e \wedge \tilde{t}_s < \tilde{t}_e$. The explicit non-empty check is necessary because ongoing time intervals can be partially empty. It is not sufficient to check if the ongoing input time intervals are not empty at all reference times; we must check non-emptiness at each reference time.

Example 2: Consider the overlaps predicate. At all reference times when one of the input time intervals instantiates to an empty time interval, the non-empty check ensures that the predicate evaluates to *false*. At all other reference times, the overlaps check determines the result. At reference time 10/16, ongoing time interval $[10/17, now)$ instantiates to an empty time interval and thus, predicate $[10/17, now)$ overlaps $[10/14, 10/20)$ evaluates to *false*. At reference time 10/18, both ongoing input time intervals instantiate to non-empty time intervals and the overlaps check evaluates to *true*. Thus, predicate $[10/17, now)$ overlaps $[10/14, 10/20)$ evaluates to ongoing boolean $\mathbf{b}[\{[10/18, \infty)\}, \{(-\infty, 10/18)\}]$.

VII. Relational Algebra

The first subsection introduces *ongoing relations* to represent query results that remain valid at varying times. Ongoing relations include tuples that belong to instantiated relations at some reference times only. An ongoing relation models this by associating each tuple with a reference time attribute. The value of the reference time attribute is restricted by the predicates on ongoing attributes. The second subsection defines the operators of the relational algebra as operators on ongoing relations.

A. Ongoing Relations

Definition 5 (Schema of an Ongoing Relation): Let $\mathbf{A}$ be a list of fixed and ongoing attributes $A_1, \ldots, A_n$ and RT be the reference time attribute. Then,

$$R = (\mathbf{A}, RT)$$

is the schema of an ongoing relation.

A tuple belongs to the instantiated relations at the reference times that are contained in the value of the tuple's reference time attribute RT. In a base tuple, the value of RT is set to trivial reference times, i.e., $RT = \{(-\infty, \infty)\}$, by the database system. The reference time of tuples is then restricted by predicates on ongoing attributes.

The bind operator $\|\mathbf{R}\|_{rt}$ instantiates an ongoing relation $\mathbf{R}$ at reference time $rt \in \mathcal{T}$ by instantiating the ongoing attributes of each tuple at reference time rt. It omits tuples whose reference time RT does not contain rt:

$$\|\mathbf{R}\|_{rt} = \{x | \exists r \in \mathbf{R}(x.\mathbf{A} = \|r.\mathbf{A}\|_{rt} \wedge rt \in r.RT)\}$$

B. Operators on Ongoing Relations

The definition of the relational algebra operators on ongoing relations follows the approach in Definition 4. For instance, selection $\sigma_\theta(\mathbf{R})$ for ongoing relations is defined as

$$\sigma_\theta(\mathbf{R}) = \mathbf{V} \text{ iff } \forall rt \in \mathcal{T}(\|\mathbf{V}\|_{rt} \equiv \sigma^F_{\theta^F}(\|\mathbf{R}\|_{rt}))$$

Derived relational algebra operators are defined as usual. As an example, $\mathbf{R} \bowtie_\theta \mathbf{S} = \sigma_\theta(\mathbf{R} \times \mathbf{S})$.

Theorem 2: Let $\mathbf{R}, \mathbf{S}$ be two ongoing relations with attributes $\mathbf{A}$ and $\mathbf{C}$, respectively. Let $\mathbf{B} \subseteq \mathbf{A}$ be a subset of the attributes of $\mathbf{R}$ and let predicate θ be composed of operations whose results remain valid as time passes by (cf. Section VI). The results of the relational algebra operators on ongoing relations are equivalent to the following ongoing relations:

Operator	Equivalence
Projection	$\pi_{\mathbf{B}}(\mathbf{R}) \equiv \{x \| \exists r \in \mathbf{R}(x.\mathbf{B} = r.\mathbf{B} \wedge x.RT = r.RT)\}$
Selection	$\sigma_\theta(\mathbf{R}) \equiv \{x \| \exists r \in \mathbf{R}(x.\mathbf{A} = r.\mathbf{A} \wedge x.RT = (r.RT \wedge \theta(r)) \wedge x.RT \neq \emptyset)\}$
Cart. prod.	$\mathbf{R} \times \mathbf{S} \equiv \{x \| \exists r \in \mathbf{R}, s \in \mathbf{S}(x.\mathbf{A} = r.\mathbf{A} \wedge x.\mathbf{C} = s.\mathbf{C} \wedge x.RT = (r.RT \wedge s.RT) \wedge x.RT \neq \emptyset)\}$
Union	$\mathbf{R} \cup \mathbf{S} \equiv \{x \| x \in \mathbf{R} \vee x \in \mathbf{S}\}$
Difference	$\mathbf{R} - \mathbf{S} \equiv \{x \| \exists r \in \mathbf{R}(x.\mathbf{A} = r.\mathbf{A} \wedge x.RT \neq \emptyset \wedge x.RT = \{rt \in r.RT \| \nexists s \in \mathbf{S}(\|r.\mathbf{A}\|_{rt} =^F \|s.\mathbf{A}\|_{rt} \wedge rt \in s.RT)\})\}$

The proof of Theorem 2 is provided in the extended online version [31].

As an example, selection $\sigma_\theta(\mathbf{R})$ selects a tuple $r \in \mathbf{R}$ by restricting the tuple's reference time RT. The reference time of the tuple is set to $r.RT \wedge \theta(r)$, i.e., the intersection of the reference time of the original tuple ($r.RT$) and the reference times when predicate $\theta(r)$ is satisfied. To restrict RT with an ongoing boolean, we convert a tuple's reference time into the set S_t of an ongoing boolean $\mathbf{b}[S_t, S_f]$ and calculate the conjunction between the ongoing booleans to determine the reference time of the result tuple.

Example 3: Consider ongoing relation $\mathbf{X}$ with tuple $x = (500, \text{Spam filter}, [01/25, now), \{(-\infty, 08/16)\})$ and selection $Q = \sigma_\theta(\mathbf{X})$ with $\theta = VT$ overlaps $[01/20, 08/18)$. Query Q selects input tuple x at the reference times when it belongs to the instantiated input relations (up to reference time 08/15) *and* when predicate $\theta(x)$ evaluates to *true*. The result of predicate $\theta(x)$ is ongoing boolean $\mathbf{b}[\{[01/26, \infty)\}, \{(-\infty, 01/26)\}]$. The reference time of result tuple y is $x.RT \wedge \theta(x)$:

$$\begin{aligned} y.RT &= x.RT \wedge \theta(x) \\ &= \{(-\infty, 08/16)\} \wedge \mathbf{b}[\{[01/26, \infty)\}, \{(-\infty, 01/26)\}] \\ &= \mathbf{b}[\{(-\infty, 08/16)\}, \{[08/16, \infty)\}] \\ &\quad \wedge \mathbf{b}[\{[01/26, \infty)\}, \{(-\infty, 01/26)\}] \\ &= \mathbf{b}[\{[01/26, 08/16)\}, \{(-\infty, 01/26), [08/16, \infty)\}] \\ &= \{[01/26, 08/16)\} \end{aligned}$$

Thus, for selection Q on input tuple x we get result tuple:

$$y = (500, \text{Spam filter}, [01/25, now), \{[01/26, 08/16)\})$$

Predicates on fixed attributes retain their standard behavior. If a predicate on fixed attributes evaluates to *true*, the result tuple's reference time does not change as it is restricted by the conjunction with ongoing boolean $\mathbf{b}[\{(-\infty, \infty)\}, \emptyset]$ ($\equiv$ *true*). If a predicate evaluates to *false*, the result tuple is omitted as the conjunction with ongoing boolean $\mathbf{b}[\emptyset, \{(-\infty, \infty)\}]$ ($\equiv$ *false*) results in an empty reference time.

VIII. Implementation

This section describes the implementation of ongoing data types in the kernel of PostgreSQL. Our implementation is space-efficient and optimized for evaluating the operations in Section VI.

Ongoing Time Data Types: Our implementation supports ongoing time points with the two granularities offered by PostgreSQL: *dates* with a granularity of days and *timestamps* with a granularity of microseconds. The PostgreSQL *date* and *timestamp* data types are extended to structures composed of two fixed dates and two fixed timestamps, respectively, to represent ongoing time points $a{+}b$. Time point now is represented as $-\infty{+}\infty$. Note that PostgreSQL natively provides representations for $-\infty$ and ∞ as fixed dates and timestamps. The extensions of the *date* and *timestamp* data types also yield support for ongoing time intervals of $\Omega \times \Omega$ as *daterange*s and *tsrange*s in PostgreSQL.

Reference Time RT: We represent a tuple's reference time as a list of fixed time intervals. For the list, we use the built-in, variable-length data type *array* to leverage the built-in storage, indexing, and fetching mechanisms for variable length data types. Its variable length guarantees that PostgreSQL allocates the minimal amount of space to store the list of reference time intervals.

Ongoing Booleans: We represent an ongoing boolean $\mathbf{b}[S_t, S_f] \in \Gamma$ with the set S_t of reference times when the ongoing boolean is *true*. S_t is represented with the same data type as a tuple's reference time. This is beneficial when restricting a tuple's reference time: the logical conjunction of a predicate and the tuple's reference time can then be directly computed (cf. Section VII-B). The time intervals used for S_t are maximal, non-overlapping, and sorted in ascending order. These properties yield an efficient implementation of the logical connectives with a sweep-line algorithm (cf. Algorithm 1).

We developed new algorithms for $<$, $\wedge$, $\vee$, and $\neg$. The less-than predicate minimizes the number of value comparisons and the implementation of the logical connectives processes each time interval just once. The other operations are implemented with the equivalences in Section VI.

Less-Than Predicate: The less-than predicate for ongoing time points is implemented according to the case distinction in Theorem 1. The result of the less-than predicate is an ongoing boolean, which we represent as an array of time intervals for S_t as described above. Since an ongoing time point $a{+}b$ ensures $a \leq b$, we use the decision tree in Fig. 6 to determine the correct case with at most three comparisons.

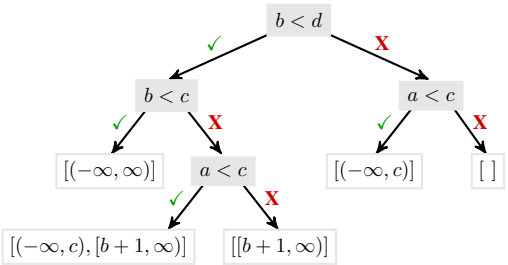


Fig. 6: Decision tree for $a{+}b < c{+}d$.

Logical Connectives: We use a sweep-line algorithm to implement the logical connectives. The implementation requires and guarantees arrays with non-overlapping time intervals that are sorted in ascending order. The implementation has the following three properties that make it efficient: 1) no sorting is required since a sweep-line algorithm guarantees sorted results at no cost, 2) each time interval of the input ongoing booleans is processed at most once, which minimizes the number of time intervals to be compared, and 3) the implementation minimizes the overall number of time point comparisons. Note that the logical connectives are not only used in predicates but also to calculate a tuple's reference time in a relational algebra operator (cf. Theorem 2). Algorithm 1 shows the implementation of the logical conjunction. The efficient implementation of the conjunction is important since the conjunction is used to calculate a result tuple's reference time in all relational algebra operators.

Procedure: Conjunction $\mathbf{b_1} \wedge \mathbf{b_2}$
Input: $\mathbf{b_1}, \mathbf{b_2} \in \Gamma$: two arrays of non-overlapping time intervals in ascending order
Output: $\mathbf{b_r} \in \Gamma$: array of non-overlapping time intervals in ascending order

```
1  b_r = [ ];  i1 ← b1.first;  i2 ← b2.first;
2  while i1 ≠ nil ∧ i2 ≠ nil do
3      if i1.te ≤ i2.ts then i1 ← b1.next;
4      else if i2.te ≤ i1.ts then i2 ← b2.next;
5      else
           // append intersection of i1 and i2
6          br.append([max(i1.ts, i2.ts), min(i1.te, i2.te)));
7          if i1.te < i2.te then i1 ← b1.next else i2 ← b2.next;
8      end
9  end
10 return br;
```

Algorithm 1: Conjunction on ongoing booleans.

Query Optimization: For the relational operators on ongoing relations, the same rules hold as for the relational algebra operators on fixed relations. For instance, the equivalence $\sigma_{\theta_1 \wedge \theta_2}(\mathbf{R}) \equiv \sigma_{\theta_1}(\sigma_{\theta_2}(\mathbf{R}))$ holds for an ongoing relation $\mathbf{R}$. After the rewriting, existing optimization techniques, such as selection push-down, join ordering, and cost-based selection of evaluation algorithms, can be used.

To leverage database optimization strategies and algorithms for queries on ongoing relations, we split a conjunctive predicate into a conjunctive predicate over fixed attributes only and a conjunctive predicate that references ongoing attributes. The predicate over fixed attributes does not depend on the reference time and can therefore be evaluated in the where clause. The predicate over ongoing attributes is used in the calculation of the result tuple's reference time (cf. Theorem 2).

IX. Evaluation

This section compares runtime, result size, and storage requirements of our solution with the state-of-the-art solution from Clifford et al. [2] and Torp et al. [3]. We vary the temporal predicate as well as the location of ongoing time intervals to evaluate their effects on runtime and result size.

A. Setup

The empirical evaluation is conducted on a 3.40 GHz machine with 16GB main memory and an SSD. The client and the database server run on the same machine. We use the

TABLE III: Characteristics of the experiment data sets.

	MozillaBugs			Incumbent	$\mathbf{D^{ex}}$	$\mathbf{D^{sh}}$	$\mathbf{D^{sc}}$
	BugInfo B	**BugAssignment A**	**BugSeverity S**				
Cardinality	394,878	582,668	434,078	83,852	10M	10M	35M
# ongoing	60,372 (15%)	63,588 (11%)	61,113 (14%)	15,805 (19%)	15%	15%	20%
Time intervals	$[a, now)$	$[a, now)$	$[a, now)$	$[a, now)$	$[a, now)$	$[now, b)$	$[a, now)$
Time span	20 years	20 years	20 years	16 years	10 years	10 years	10 years

PostgreSQL 9.4.0 kernel extended with our implementation of ongoing data types and the operations on them.

Table III summarizes the real-world and synthetic data sets. As ongoing time intervals we use expanding time intervals $[a, now)$ and shrinking time intervals $[now, b)$. Note that the duration of expanding ongoing time intervals increases as the reference time increases. The earlier an expanding time interval starts, the more time intervals it overlaps with. We use the real-world data sets *MozillaBugs* [32] and *Incumbent* [33]. The *MozillaBugs* data set records the history of bugs in the Mozilla project. It contains the following three relations. (1) **BugInfo** records general information about a bug: ID, product, component, operating system, textual description, and valid time. Bugs that have not been resolved as of the date of the data export have ongoing valid time intervals. (2) **BugAssignment** records the email address of the person assigned to a bug, the bug id, and the valid time. (3) **BugSeverity** records the bug id, the severity of the bug, and the valid time. The last assignment and last severity of bugs with ongoing valid times have ongoing valid times as well. *Incumbent* records the valid time periods during which projects are assigned to university employees. We converted project assignments that were not finished at the date of the data export into tuples with ongoing assignments, resulting in 19% ongoing tuples.

Fig. 7 shows the distribution of the start points of the ongoing time intervals. In *MozillaBugs*, 50% of the tuples with ongoing time intervals in relations **BugInfo**, **BugAssignment**, and **BugSeverity** are located within the last two years of the history. In *Incumbent*, all ongoing project assignments started within the last year of the history. For experiments with an increasing number of tuples we grow the size of the real-world data sets by growing the history backward. This means that the percentage of ongoing time intervals decreases as the data size grows. For *MozillaBugs*, we grow the history backward for the **BugInfo** relation and use all records in the other two relations that match to the bug ids in **BugInfo**.

To maximize performance we implemented the bind operator of Clifford et al. [2] in the PostgreSQL 9.4.0 kernel as a C function that is called when an ongoing attribute is accessed [3]. $\text{Cliff}_{\max}$ refers to Clifford's approach that uses a reference time that is greater than the latest end point. It represents the typical use case with reference times close to the current time.

We use two relational algebra operators for the evaluation: selection $Q_i^{\sigma} = \sigma_{VT\ \text{pred}_i\ [t_s,t_e)}(\mathbf{R})$ with a temporal predicate on the valid time and join $Q_i^{\bowtie} = \mathbf{R} \bowtie_{\theta_N \wedge\ \mathbf{R}.VT\ \text{pred}_i\ \mathbf{S}.VT} \mathbf{S}$ whose join predicate includes equality predicates on non-temporal attributes (θ_N) and a temporal predicate pred_i on the valid time. $\mathbf{S}$ and $\mathbf{R}$ refer to the same relation. The fixed time

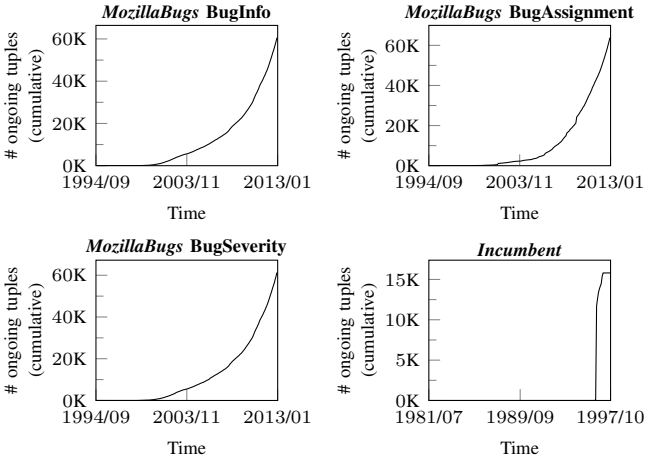


Fig. 7: Start point distribution of ongoing intervals.

interval $[t_s, t_e)$ in the selection predicate spans the last 10% of the data history. Selection is a fast operator and will show the overhead of our approach; join queries are common for database systems and representative for different workloads. On *MozillaBugs*, we use a complex join query to evaluate our approach on a heavier workload as well. The join query determines for a person similar bugs that are open at any time when the person is working on a bug with severity *major*. Similar bugs are bugs that affect the same product, component, and operating system (θ_{sim}):

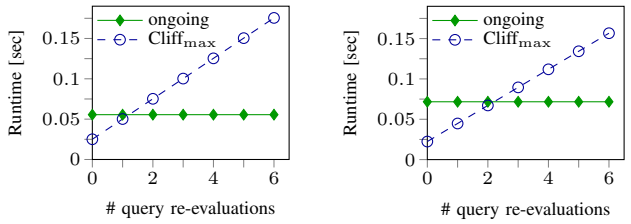
$$QC_i^{\bowtie} = \mathbf{A} \bowtie_{\mathbf{A}.ID=\mathbf{S}.ID \wedge \mathbf{A}.VT \text{ overlaps } \mathbf{S}.VT \wedge \text{Severity='major'}} \mathbf{S} \bowtie_{\mathbf{A}.ID=\mathbf{B}.ID} \mathbf{B} \bowtie_{\theta_{sim} \wedge \mathbf{A}.VT\ \text{pred}_i\ \mathbf{B}'.VT} \mathbf{B}'$$

As temporal predicates, we use *overlaps* ($\text{pred}_{\text{ovlp}}$) and *before* ($\text{pred}_{\text{bef}}$). These predicates are representative for the most commonly used temporal predicates [34]–[38]. The ongoing approach uses the predicates for ongoing time intervals (cf. Section VI). To maximize the performance of Clifford's approach, we use the predicates for fixed time intervals.

B. Query Re-Evaluations

Our approach evaluates a query to an *ongoing result* that is returned to an application. Since ongoing results do not get invalidated by time passing by, the application does not have to re-evaluate the query. In contrast, Clifford's query results get invalidated as time passes by and thus, the application must re-evaluate the query. First, we evaluate the break-even point of the ongoing approach for different predicates. Next, we evaluate the impact of the location and number of ongoing time intervals on the runtime.

Number of Query Re-Evaluations: The ongoing approach has a runtime overhead due to the handling of the predicates on ongoing time points and time intervals and due to possibly larger result sizes (cf. Section IX-D). This is shown in Fig. 8 on the real world data *Incumbent* for the temporal predicates *overlaps* and *before*. Clearly, the ongoing approach already performs better after very few query re-evaluations. Specifically, the ongoing approach is faster after two re-evaluations for the *overlaps* predicate (Fig. 8a) and after three re-evaluations for the *before* predicate (Fig. 8b). Selection Q^{σ}_{ovlp} is faster than selection Q^{σ}_{bef} for ongoing time intervals because the optimized implementation of the *overlaps* predicate requires about half as many fixed-value comparisons per tuple as the *before* predicate.



(a) Q^{σ}_{ovlp} with *overlaps*. (b) Q^{σ}_{bef} with *before*.

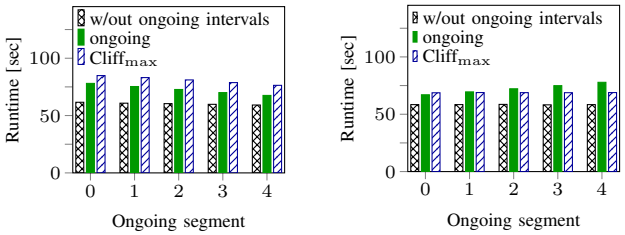
Fig. 8: Number of query re-evaluations on *Incumbent*.

Location of Ongoing Time Intervals: We vary the location of the ongoing time intervals by dividing the 10 year history into 5 segments (2 years each) and placing all start points (D^{ex}) or end points (D^{sh}) of the ongoing intervals into one of the segments. Ongoing segment 0 spans the first two years. Fig. 9 shows the impact of the location on the runtime for one re-evaluation. Since D^{ex} contains expanding ongoing time intervals, the runtime of the ongoing approach decreases for the *overlaps* predicate if the ongoing time intervals are placed in the later segments (cf. Fig. 9a). Fig. 9b shows that the opposite observation holds for shrinking ongoing time intervals in D^{sh} since their duration is longer when their end points are placed in later ongoing segments. To establish a baseline for the runtime, we replaced all ongoing time intervals in the two datasets with fixed time intervals and evaluated query $Q^{\bowtie}_{\text{ovlp}}$ on these data sets (without ongoing time intervals). Observe that the baseline runtime accounts for 80% to 90% of the runtime of the ongoing approach. Thus, the join processing is the expensive part and the runtime overhead for processing ongoing time intervals is less than 20%.

Number of Input Tuples: We evaluate the scalability by increasing the size of the input relation. Fig. 10a shows that the ongoing approach has a similar linear runtime increase as Clifford's approach does with increasing input sizes. Thus, as shown in Fig. 10b, the number of query re-evaluations after which the ongoing approach performs better stays constant as the number of input tuples increases.

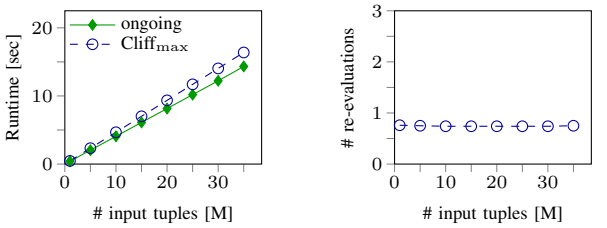
C. Instantiated Query Results via Materialized Views

Ongoing relations can easily be combined with materialized views to efficiently compute instantiated results at different



(a) $Q^{\bowtie}_{\text{ovlp}}$ on D^{ex}. (b) $Q^{\bowtie}_{\text{ovlp}}$ on D^{sh}.

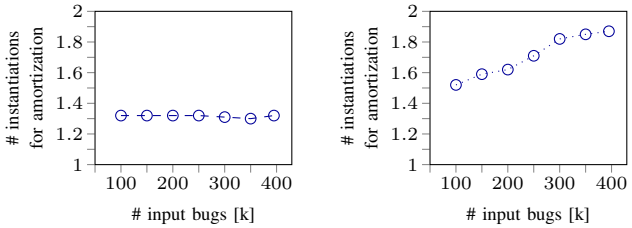
Fig. 9: Location of ongoing time intervals.



(a) Runtime. (b) Query re-evaluations.

Fig. 10: Number of input tuples (Q^{σ}_{ovlp} on D^{sc}).

reference times. This allows applications that do not want to handle ongoing relations explicitly to leverage the performance benefits of ongoing relations. We evaluate the runtime amortization of the ongoing approach, i.e., at how many different reference times n an instantiated result must be returned to an application, such that calculating the ongoing result and instantiating it at the n reference times outperforms Clifford's approach, which must calculate the query at each of the n reference times. The main factors for the amortization are (1) the complexity of the query and (2) the reference time used for the instantiation.



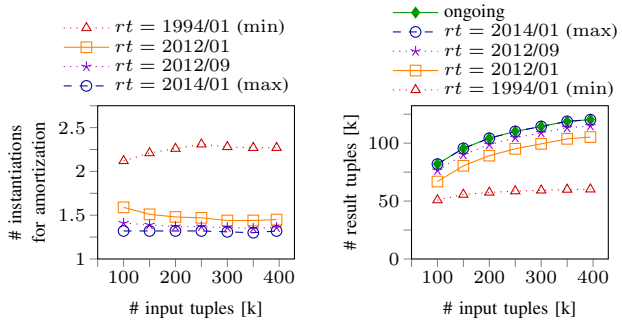
(a) Selection $Q^{\sigma}_{\text{ovlp}}(\mathbf{B})$. (b) Join $QC^{\bowtie}_{\text{ovlp}}(\mathbf{A}, \mathbf{S}, \mathbf{B})$.

Fig. 11: Amortization for selection and join on *MozillaBugs*.

Query Complexity: Fig. 11 shows the amortization for selection and complex join. The number of input bugs (x-axis) is equal to the number of tuples in relation $\mathbf{B}$ (cf. Section IX-A on how we vary the size of the dataset). Both queries require less than two instantiations for the amortization at all input sizes. For the selection query, the number of reference times for amortization remains constant with varying input size. For the complex join, it increases slightly: around 25% for a 300% input bugs increase. This is because the query optimizer chooses a linear-time hash join for Clifford's approach when evaluating the join with $\mathbf{B}'$, whereas it uses a log-linear-

time merge join for the ongoing approach. This additional logarithmic component is consistent with the curve in Fig. 11b.

Reference Time: Smaller size differences of the ongoing and instantiated query result lead to a faster runtime amortization of the ongoing approach. The size of the ongoing result is independent of the reference time whereas the size of the instantiated result depends on it. Fig. 12a shows that the amortization of the ongoing approach decreases from three instantiations for early reference times ($rt = \min$, i.e., smallest time point in the data set) to two instantiations for later reference times. For the *overlaps* predicate, later reference times result in smaller size differences: the later the reference time, the more ongoing time intervals instantiate to non-empty time intervals. Thus, more and more ongoing time intervals satisfy the predicate (especially as a late selection time interval is used) and belong to the result (Fig. 12b).



(a) Amortization. (b) Result size.

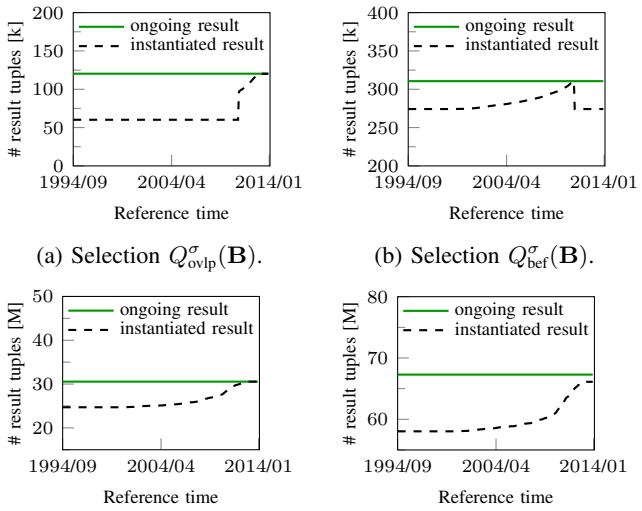
Fig. 12: Amortization for $Q^{\sigma}_{\text{ovlp}}(\mathbf{B})$ on *MozillaBugs*.

D. Storage

The ongoing approach requires additional storage for each tuple and for the tuples that belong to the ongoing result but not to Clifford's result. The per-tuple storage overhead is the additional RT attribute and a doubling of the size of the valid time attribute (because ongoing rather than fixed values are used). Typically, the value of the RT attribute can be represented with one fixed time interval. The details, along with an empirical evaluation, are discussed in the extended online version [31].

The number of additional tuples that are part of the ongoing result but not of Clifford's result depends on the reference time. Since ongoing results combine the results at all reference times, they must contain at least the tuples of the largest instantiated result. If the size of the ongoing result and the largest instantiated result are equal, the size of the ongoing result is optimal.

For expanding ongoing intervals the size of the ongoing result is optimal for predicate *overlaps* (Fig. 13a and Fig. 13c). As the duration of expanding time intervals increases, once an expanding time interval overlaps with a time interval, they remain overlapping for all reference times afterwards. Tuples are only added to the instantiated query results with increasing reference times and thus, the ongoing result contains exactly the tuples of the largest instantiated result.



(a) Selection $Q^{\sigma}_{\text{ovlp}}(\mathbf{B})$. (b) Selection $Q^{\sigma}_{\text{bef}}(\mathbf{B})$.

(c) Join $QC^{\bowtie}_{\text{ovlp}}(\mathbf{A},\mathbf{S},\mathbf{B})$. (d) Join $QC^{\bowtie}_{\text{bef}}(\mathbf{A},\mathbf{S},\mathbf{B})$.

Fig. 13: Result size vs. reference time on *MozillaBugs*.

For expanding ongoing intervals and the *before* predicate, the ongoing result reaches the optimal size for selections (Fig. 13b) and gets close to it for joins (Fig. 13d). Due to the duration increase, expanding ongoing time intervals are before a time interval up to a reference time and then stop being before it. As there is one selection interval in the selection, this reference time is the same for all expanding time intervals (it is the start point of the selection interval). In a join, an expanding time interval is compared to multiple time intervals. Usually there does not exist a single reference time that belongs to the RT attribute of all result tuples, and thus, the maximum instantiated result is smaller than the ongoing result.

E. Summary

As expected, the ongoing approach has a runtime overhead to compute ongoing results that do not get invalidated by time passing by. This overhead is quite small and pays off for as little as three query re-evaluations of Clifford's approach when returning an ongoing result and for returning as little as two instantiated results when leveraging the ongoing result to calculate them. For late reference times, which are close to the current time, the result size of the ongoing approach is equal to the result size of Clifford's approach for the widely-used *overlaps* predicate and close to equal for other predicates. Thus, the number of tuples that are contained in an ongoing result but not in Clifford's result is small.

X. Conclusions

We propose the first approach that evaluates queries on ongoing relations without instantiating ongoing time points. Ongoing time points are preserved in query results and the results remain valid as time passes by. For database systems this is a crucial property as it guarantees that cached results, materialized views, etc. have to be maintained only after explicit database modifications. We define predicates and functions on ongoing time points and time intervals.

We propose *ongoing relations* that associate each tuple with a reference time attribute. The value of the reference time attribute contains the reference times when a tuple belongs to the instantiated relations and is restricted by predicates on ongoing attributes.

There are several interesting topics for future research. First, we want to extend the set of functions for ongoing data types to include a duration function for ongoing time intervals whose result are ongoing integers. Second, we plan to propose an aggregation operator for ongoing relations and determine the additional ongoing data types that are required to support aggregation and group tuples in the presence of RT and ongoing attributes. Finally, we want to develop index access methods for ongoing time points (based on the approaches for indexing fixed time intervals) and discuss query classes that benefit from these indexes.

REFERENCES

[1] C. S. Jensen and R. T. Snodgrass, "Valid-Time and Transaction-Time Relation," in *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu, Eds. Springer Publishing Company, Incorporated, 2009, pp. 3254–3254.

[2] J. Clifford, C. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass, "On the Semantics of Now in Databases," *ACM Transactions on Database Systems (TODS)*, vol. 22, no. 2, pp. 171–214, 1997.

[3] K. Torp, C. S. Jensen, and R. T. Snodgrass, "Modification Semantics in Now-Relative Databases," *Information Systems*, vol. 29, no. 8, pp. 653–683, Dec. 2004.

[4] L. Anselma, B. Stantic, P. Terenziani, and A. Sattar, "Querying Now-Relative Data," *Journal of Intelligent Information Systems*, vol. 41, no. 2, pp. 285–311, 2013.

[5] J. Melton and A. R. Simon, *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, 1993, pages 69,107,459.

[6] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass, Eds., *Temporal Databases: Theory, Design, and Implementation*. Benjamin-Cummings Publishing Co., Inc., 1993.

[7] M. D. Soo, R. T. Snodgrass, and C. S. Jensen, "Efficient Evaluation of the Valid-Time Natural Join," in *Proceedings of the 10th International Conference on Data Engineering*. IEEE, 1994, pp. 282–292.

[8] I. F. V. Lopez, R. T. Snodgrass, and B. Moon, "Spatiotemporal Aggregate Computation: A Survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 2, pp. 271–286, 2005.

[9] M. H. Böhlen, J. Gamper, and C. S. Jensen, "Multi-Dimensional Aggregation for Temporal Data," in *Proceedings of the 10th International Conference on Advances in Database Technology*, ser. EDBT'06, 2006, pp. 257–275.

[10] J. Bair, M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass, "Notions of Upward Compatibility of Temporal Query Languages," *Wirtschaftsinformatik*, vol. 39, no. 1, pp. 25–34, 1997.

[11] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass, "Temporal Statement Modifiers," *ACM Transactions on Database Systems (TODS)*, vol. 25, no. 4, pp. 407–456, 2000.

[12] A. Dignös, M. H. Böhlen, J. Gamper, and C. S. Jensen, "Extending the Kernel of a Relational DBMS with Comprehensive Support for Sequenced Temporal Queries," *ACM Transactions on Database Systems (TODS)*, vol. 41, no. 4, pp. 26:1–26:46, Nov. 2016.

[13] K. Torp, C. S. Jensen, and R. T. Snodgrass, "Effective Timestamping in Databases," *The VLDB Journal*, vol. 8, no. 3-4, pp. 267–288, 2000.

[14] M. Finger and P. McBrien, "On the Semantics of Current-Time in Temporal Databases," in *Proceedings of the 11th Brazilian Symposium on Databases*, 1996, pp. 324–337.

[15] C. S. Jensen and D. B. Lomet, "Transaction Timestamping in (Temporal) Databases," in *Proceedings of the 27th International Conference on Very Large Data Bases*, ser. VLDB, 2001, pp. 441–450.

[16] C. E. Dyreson, C. S. Jensen, and R. T. Snodgrass, "Now in Temporal Databases," in *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu, Eds. Boston, MA: Springer US, 2009, pp. 1920–1924.

[17] K. Torp, C. S. Jensen, and M. H. Böhlen, "Layered Temporal DBMS's: Concepts and Techniques." in *Proceedings of the Fifth International Conference on Database Systems for Advanced Applications (DASFAA)*. World Scientific Press, 1997, pp. 371–380.

[18] B. Stantic, J. Thornton, and A. Sattar, "A Novel Approach to Model NOW in Temporal Databases," in *Proceedings of the 10th International Symposium on Temporal Representation and Reasoning and Fourth International Conference on Temporal Logic.*, 2003, pp. 174–180.

[19] B. Stantic, A. Sattar, and P. Terenziani, "The POINT Approach to Represent Now in Bitemporal Databases," *Journal of Intelligent Information Systems*, vol. 32, no. 3, pp. 297–323, Jun. 2009.

[20] L. Anselma, L. Piovesan, A. Sattar, B. Stantic, and P. Terenziani, "A Comprehensive Approach to Now in Temporal Relational Databases: Semantics and Representation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 10, pp. 2538–2551, Oct 2016.

[21] R. T. Snodgrass, "The Temporal Query Language TQuel," *ACM Transactions on Database Systems (TODS)*, vol. 12, no. 2, pp. 247–298, 1987.

[22] R. H. Güting and M. Schneider, *Moving Objects Databases*. Elsevier, 2005.

[23] S. Chen, B. C. Ooi, K.-L. Tan, and M. A. Nascimento, "ST2B-tree: A Self-Tunable Spatio-Temporal B+-tree Index for Moving Objects," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. ACM, 2008, pp. 29–42.

[24] K. Tzoumas, M. L. Yiu, and C. S. Jensen, "Workload-Aware Indexing of Continuously Moving Objects," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 1186–1197, Aug. 2009.

[25] R. Zhang, J. Qi, D. Lin, W. Wang, and R. C.-W. Wong, "A Highly Optimized Algorithm for Continuous Intersection Join Queries over Moving Objects," *The VLDB Journal*, vol. 21, no. 4, pp. 561–586, Aug. 2012.

[26] S. Nutanong, M. E. Ali, E. Tanin, and K. Mouratidis, "Dynamic Nearest Neighbor Queries in Euclidean Space," in *Encyclopedia of GIS*, S. Shekhar, H. Xiong, and X. Zhou, Eds. Springer International Publishing, 2015, pp. 1–7.

[27] J. Lim, Y. Lee, K. Bok, and J. Yoo, "A Continuous Reverse Skyline Query Processing for Moving Objects," in *2014 International Conference on Big Data and Smart Computing (BIGCOMP)*, Jan 2014, pp. 66–71.

[28] H. Hu, J. Xu, and D. L. Lee, "A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. ACM, 2005, pp. 479–490.

[29] Y. Tao and D. Papadias, "Time-Parameterized Queries in Spatio-Temporal Databases," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. ACM, 2002, pp. 334–345.

[30] C. S. Jensen, J. Clifford, S. K. Gadia, A. Segev, and R. T. Snodgrass, "A Glossary of Temporal Database Concepts," *SIGMOD Record*, vol. 21, no. 3, pp. 35–43, Sep. 1992.

[31] Y. Mülle and M. H. Böhlen, "Query Results over Ongoing Databases that Remain Valid as Time Passes By (Extended Version)," *Technical Report CoRR*, 2020. [Online]. Available: https://arxiv.org/pdf/2001.05722.pdf

[32] A. Lamkanfi, J. Pérez, and S. Demeyer, "The Eclipse and Mozilla Defect Tracking Dataset: A Genuine Dataset for Mining Bug Information," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 203–206.

[33] J. A. G. Gendrano, R. R. Shah, R. T. Snodgrass, and J. Yang, "University Information System (UIS) Dataset. TimeCenter CD-1," 1998.

[34] TPC. (2017) TPC Transaction Processing Performance Council, TPC-H. Retrieved October 2017 from http://www.tpc.org/tpch. [Online]. Available: http://www.tpc.org/tpch

[35] A. Dignös, M. H. Böhlen, and J. Gamper, "Overlap Interval Partition Join," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, 2014, pp. 1459–1470.

[36] F. Cafagna and M. H. Böhlen, "Disjoint Interval Partitioning," *The VLDB Journal*, vol. 26, no. 3, pp. 447–466, Jun. 2017.

[37] P. Bouros and N. Mamoulis, "A Forward Scan Based Plane Sweep Algorithm for Parallel Interval Joins," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1346–1357, Aug. 2017.

[38] D. Piatov, S. Helmer, and A. Dignös, "An Interval Join Optimized for Modern Hardware," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, May 2016, pp. 1098–1109.